Active Measurement of the Impact of Network Switch Utilization on Application Performance

Marc Casas
Barcelona Supercomputing Center
Jordi Girona, 29. Nexus II Building
08034 Barcelona

Greg Bronevetsky
Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA, 94550

***Abstract*—Inter-node networks are a key capability of High-Performance Computing (HPC) systems that differentiates them from less capable classes of machines. However, in spite of their very high performance, the increasing computational power of HPC compute nodes and the associated rise in application communication needs make network performance a common performance bottleneck. To achieve high performance in spite of network limitations application developers require tools to measure their applications' network utilization and inform them about how the network's communication capacity relates to the performance of their applications.**

This paper presents a new performance measurement and analysis methodology based on empirical measurements of network behavior. Our approach uses two benchmarks that inject extra network communication. The first probes the fraction of the network that is utilized by a software component (an application or an individual task) to determine the existence and severity of network contention. The second aggressively injects network traffic while a software component runs to evaluate its performance on less capable networks or when it shares the network with other software components. We then combine the information from the two types of experiments to predict the performance slowdown experienced by multiple software components (e.g. multiple processes of a single MPI application) when they share a single network. Our methodology is applied to individual network switches and demonstrated taking 6 representative HPC applications and predicting the performance slowdowns of the 36 possible application pairs. The average error of our predictions is less than 10%.

I. Introduction

HPC applications demand very capable communication networks to support their high message and data volumes and/or tight synchronizations. Indeed, constraints on available network bandwidth or latency as well as network hotspots induced by specific communication patterns are often the key bottleneck that limit application performance [24], [2]. Looking into the future, it is expected that the computational capabilities of individual computing nodes will continue to rise faster than the capabilities of the networks that connect them [16]. This means that application performance will become increasingly bottlenecked on the capabilities of the network, making it even more imperative for application developers to optimize their applications taking network performance into account. Specifically, developers will need to (i) predict how their applications will perform on future systems with poorer network-to-node performance ratios and (ii) develop ways to assign computing work to available resources to effectively balance network communication and on-node computation. To achieve these tasks developers will require powerful tools to enable them to understand the interactions between their applications and the networks they run on and how these interactions ultimately affect application performance. Specifically, two directions of this interaction will need to be quantified for developers. First, tools must quantify how the application's communication utilizes the network and whether the application's needs are approaching the limits of the network's capabilities. Second, tools must measure how the capabilities of the network influence application performance and most importantly, whether the network is the application's performance bottleneck. These analyses must apply to both current and future systems, as well as to both static and highly configurable applications (e.g. where the space of possible configurations is too large to be explicitly enumerated and analyzed).

This paper presents a new approach to measure the relationship between network capability and application performance. Our basic insight is that this relationship should be modeled as the application consuming a resource provided by the network. As more of this resource is available, the application runs monotonically faster, with reduced improvements as application performance becomes bottlenecked on other resources. Further, if multiple software components (entire applications or individual tasks such as processes, threads or Charm++ chares [13])) run concurrently on the same network, they will share its resources. This sharing can be modeled as one component consuming some amount of network resources, making it unavailable to others and thus causing them to behave as if they were running on a less capable network. The heart of this idea is a "performance relativity" principle, that "from the perspective of software components less capable

The research leading to these results has received funding from the European Research Council under the European Union's 7th FP (FP/2007-2013) / ERC GA n. 321253. Work partially supported by the Spanish Ministry of Science and Innovation (TIN2012-34557). This article has been authored in part by Lawrence Livermore National Security, LLC under Contract DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. This work was partially supported by the Department of Energy Office of Science (Advanced Scientific Computing Research) Early Career Grant, award number NA27344.

networks behave very similarly to networks that are partially utilized by other software components". This principle enables two novel measurement techniques that can answer the above questions:

Impact experiments measure a software component's use of the network based on the latency of a few additional packets sent over the network while the component runs. These measurements directly quantify the network's ability to carry application communication and can be used to determine whether the network is congested and measure how close the application is to fully utilizing the network. The additional packets are triggered by extra tasks running on dedicated cores and they do not impact applications' performance as the extra load is very low.

Compression experiments measure the relationship between network capability and a software component's performance. The component is executed concurrently with a micro-benchmark that runs on cores connected to the same network and sends varying volumes of communication. As the effective network capability is varied we observe the component's resulting performance, which corresponds to how it will perform on less capable networks or when more software components are executed on the same network.

Finally we present several techniques that combine the two measurements to predict the performance degradation that a given combination of software components would suffer when executed concurrently on the same network. Each technique is based on a particular description of the available network capability when an application is running. Data from Impact measurements is used to compute latencies of the triggered packets. We consider four different approaches to describe the available network capability when a particular parallel application is running: i) The average latency of all the packets triggered by the application ii) The average latency and the standard deviation of the packets triggered by the application iii) the histogram of the latencies of the packets triggered and iv) a mathematical queue [27]. By measuring the network capability that is left available while a given application or the Compression benchmark runs we can estimate the effect of multiple concurrent software components on each other as they share a network. The experimental and analytic procedures presented in this paper are focused on single-switch networks that connect multiple computing nodes.

Our approach improves upon the state of the art in network performance modeling and measurement in the following ways:

i) Impact experiments of network utilization and contention are significantly faster than similar analyses performed inside simulators and apply to real physical networks for which precise models may not exist due to intellectual property restrictions. Further, unlike indirect measurement techniques, Impact experiments directly probe the network's ability to carry out the application's communication requests. Since they focus on just the network and quantify its effective capabilities in terms of a generic queue-oriented metric, these experiments provide a simple and unfiltered view onto this resource.

ii) Compression experiments and Performance Degradation analysis make it possible to relate application performance to network capability. While simulators can predict the performance of specific workloads on specific networks, a separate simulation run is required for each configuration. As the number of configuration options increases (e.g. number of atoms per core or the assignment of software components to different cores), the number of such experiments rises exponentially. In contrast, our approach scales linearly with the number of software components that must be measured independently.

iii) Our techniques are enabled by several new metrics for measuring network utilization. These metrics have the key property of describing the utilization of the physical network as well as the resulting application performance.

This paper is structured as follows: Section II presents the experimental setup used and briefly describes the set of applications we use in our experiments. Section III-A describes Impact measurements and how they can be used to feed an analytical model based on queuing theory. Section III-B describes Compression measurements, details how they Interact with impact measurements and shows the performance analyses they enable for real applications. Section V presents and validates our methodology for predicting performance of real complex workloads that share the same network switch.

II. Experimental design

The experiments in this paper were conducted on the Cab cluster at the Lawrence Livermore National Laboratory. Cab is composed of 1,296 compute nodes, each of which includes two 8-core 2.6Ghz Intel Xeon E5-2670 processors with 32GB of RAM. The network is QLogic Quad-data-rate organized into a two-level fat tree. This paper focuses on the bottom-level switches of the network, which are QLogic 12300, with 36 ports, of which 18 are used to connect compute nodes and 18 more connect to the second-level switch. These switches provide approximately 1μs of network latency and 5GB/s bandwidth. Each experiment on Cab was run on groups of 18 nodes, respectively, connected to a single bottom-level switch and our results are thus not affected by interference from other applications running on the same cluster.

Our experiments focus on the following applications:

- AMG [10] - An implementation of the Algebraic Multi Grid Solver by using the Hypre library.
- FFTW [11] - Fast Fourier Transform library that uses hierarchical composition of multiple FFT algorithms, applied to perform a 2D transform of a 2000x2000 matrix.
- Lulesh [1] - The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics simulation that is a materials science proxy application, executed on a 22x22x22 cube domain.
- MCB [7] - A continuous energy Monte Carlo Burnup Simulation Code for studying nuclear waste transmutation systems, executed on 3,000,000 particles.
- MILC [3] - The MIMD Lattice Computation, a Quantum Chromodynamics simulation with lattice size nx=16,

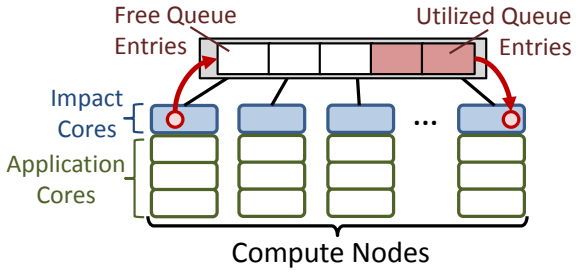

Fig. 1. Impact Interference

ny=32, nz=32, nt=36.

- VPFFT [19] -A structure sensitive crystal plasticity simulation code.

The set of applications is representative of the typical workloads that run in HPC infrastructures. AMG carries out several iterations of an iterative solver over the same linear system at different levels of granularity, which means that it behaves like a CPU intensive benchmark when it operates over a dense representation of the system and like a communication and memory bound application when it performs solver iterations over a sparse representation of the system. Thus, AMG runs will display very different phases. FFTW and VPFFT applications contain expensive all-to-all communications. The difference between these two applications is that VPFFT performs expensive computation between two communication phases while FFTW does not. As such, VPFFT has some flexibility to overlap communication and computation while FFTW has much less. Lulesh is a typical finite difference method code with local communication phases interleaved by intensive computation phases. MCB is a monte carlo simulation code, which means that it does not have much communication and, therefore, its usage of the interconnecting network is expected to be low. Finally, MILC spends most of its time running the conjugate gradient solver, which means that most of its communications involve point to point communications with the neighbors and global reductions once in a while.

III. Active Measurement

Our active measurement methodology adds extra-load into the network and measures some performance metrics provided, directly or indirectly, by this extra load. We follow two main approaches: The first one aims to inject a very light extra traffic into the network with the aim of not impacting the performance of the running application. By directly measuring the latency of the extra packets we inject, we infer the distribution of the latencies of the packets triggered by the main application, which would be very hard to measure without injecting the extra traffic. The second approach aims to inject heavy traffic into the network and measure, for each degree of interference, the performance degradation suffered by the main application. We explain the details of these two approaches in this section.

```
while(1) {
  if( my_node%2 == 0 && my_node!=n_nodes-1 ) {
        MPI_Isend(... , (my_rank+tasks_per_node)%(n_nodes), ... , &rqt);
        MPI_Irecv(... , (my_rank+tasks_per_node)%(n_nodes), ... , &rqt2);
  } else if ( my_node%2 == 1 ) {
        MPI_Irecv(... , my_rank-tasks_per_node , ... , &rqt);
        MPI_Isend(... , my_rank-tasks_per_node , ... , &rqt2);
   }
   MPI_Wait(&request , status);
   MPI_Wait(&request2 , status);
   usleep(100000);
}
```

Fig. 2. Pseudo-code of the ImpactB micro-benchmark

A. Impact

The basic idea behind Impact experiments is that the degree to which an application utilizes a network switch can be measured in terms of how well the network can service additional communication requests. Application messages are broken up into multiple small (few KB) packets and sent to the network switch. As illustrated in Figure 1, packets from one compute node arrive on one port of the switch, propagate through its internal circuitry and exit via the port of its destination node. Since the execution time of communication operations depends on the transit time of each packet, the distribution of these times captures the network's effective capability that is available to applications. Further, when some software component is already utilizing the network, the difference between this distribution during the component's execution and the same on an unloaded network measures the amount of network capability the component uses up and leaves unavailable to others.

We measure the latency of packets through the network switch using the simple micro-benchmark listed in Figure 2, which we denote ImpactB. Compute nodes on the same switch are paired and execute a ping-pong data exchange where the process with the even rank sends a message, the process with the odd rank receives it and replies with another, which is finally received by the initial process. The entire exchange is timed by the initiator process to determine the average latency of the two messages, which are set to be 1KB in size to ensure that they are communicated via a single network packet. Each ping-pong exchange is separated by a 100ms sleep to minimize ImpactB's effect on the executing application.

Figure 3 shows the distribution of message latencies observed on Cab when executing ImpactB on an unloaded switch and when ImpactB is executed concurrently with our target applications. In these experiments the processes of ImpactB and the target application were spread over all the compute nodes connected to the switch. On our experiments, 2 ImpactB processes was executed on every node. Since Cab's nodes have 2 sockets, an ImpactB process was run on each socket.

The application processes were executed on the remaining cores. We executed 4 processes of MILC, FFTW, MCB, VPFFT and AMG on each socket, 8 per node for a total of 144 across all the 18 nodes connected to a switch. Lulesh, which needs a cubic number of processes, was run on 16 nodes, utilizing 2 cores on each socket, for a total of 64 MPI processes.

The remaining cores were left idle in these experiments.

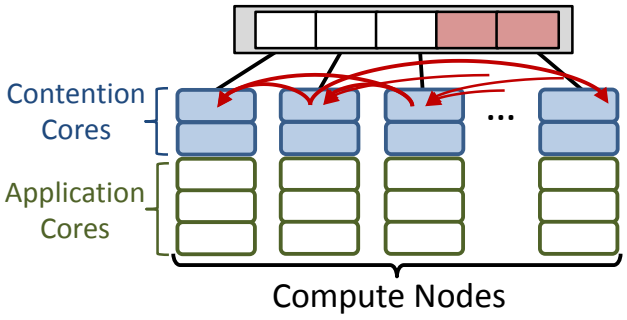

Fig. 4. Interference by the CompressionB micro-benchmark

This assignment of application processes to cores was used to simplify the presentation of the performance prediction experiments in Section V, which discusses performance prediction for multiple concurrently-executing applications.

The data shows that when the switch is not loaded, packet latency is 1.25μs on average, with many packets taking a little less or more time and a few packets taking significantly longer. When the applications are running the latency distribution shifts. The execution of FFTW and MCB on Cab shifts 20% of packets from taking approximately 1.25μs to take more than 2.5μs. In contrast, the primary effect of Lulesh and MILC is to shift the mode of the distribution to the right, close to 2.5μs. Further, while Lulesh didn't cause an increase in the fraction of packets with very high latency, with MCB this effect was strong.

As we show in Section V, by using these direct measurements of application behavior on the network switches we can make quantitative prediction of application slowdown in different utilization scenarios without knowing anything about the internal details of the switches or the applications that use them.

B. Compression

Compression experiments measure the relationship between the network capability available to a software component and its performance by incrementally reducing network capability and observing the effect of this on performance. Since it is not possible to adjust the properties of real switches and network simulations are expensive, we use the performance relativity principle (reduced network capability affects application performance similarly to resource sharing) to simulate reduced network capability via software interference. We execute the target software component on a subset of the available cores. On the remaining cores we execute the CompressionB micro-benchmark, the pseudo-code for which is listed in Figure 5. CompressionB is executed on the same number of cores on each node, where processes running on the same core ID on different nodes are organized in a 1-dimensional communication ring. As illustrated in Figure 4, in each iteration every CompressionB process sends a 40KB message to P partner processes that precede it in the ring (all processes in its ring are on different nodes) and receives the messages sent by the P succeeding processes. After M messages have been sent in this way, the benchmark sleeps for B cycles, waits for all the `MPI_Irecvs` and `MPI_Isends` to complete, and repeats the communication pattern.

```
while(1) {
  for(partner=0; partner<P; partner++) {
    for(mesg=0; mesg<M; mesg++) {
      // Receive from same core ID on succeeding node
      MPI_Irecv( ... , (my_rank+tasks_per_node*(partner+1))%comm_size, ... );
      // Send to same core ID on the preceding node
      MPI_Isend( ... , (my_rank-tasks_per_node*(partner+1)+comm_size)%comm_size, );
    }
    usleep(B);
  }
  MPI_Waitall( ... );
}
```

Fig. 5. Pseudo-code of the CompressionB interference micro-benchmark

Various settings of parameters P, M and B degrade network capability to different extents. Thus, by performing multiple experiments where a different configuration of CompressionB is executed concurrently with a target software component it is possible to measure the degradation in the component's performance on less capable switches. This corresponds to future systems where the network performance is poorer relative to processor performance, as well as scenarios where more application work is assigned to the same network.

IV. Modeling

In this section we describe the four modeling approaches we follow to get slowdown predictions when applications share network resources. These four approaches can be divided into two main categories: The look-up table based models and the queue model. Three of our four approaches are look-up table based while just the fourth is the queue approach. All of them use information obtained from the impact and compression measurement to compute the performance slowdown predictions.

A. Look-up Table Models

The look-up table models use a description of the intensity of the extra traffic injected by the compression benchmark and the performance degradation each application suffers for each level of traffic injection. As the compression benchmark has many different input configurations, we can consider from very light-weight to heavy traffic injections, which describes the application behavior under very different contexts. Additionally, the degree of perturbation each application brings, measured by using the impact benchmark, is also used.

To predict the performance slowdown of a particular application when it shares the network switch with a second workload, the model takes the level of perturbation that the second application brings, which is measured by the impact benchmark and summarized in a certain way, looks into a look-up table for the input configuration of the compression benchmark that brings the closest degree of perturbation, and then takes the subsequent performance degradation, previously measured by the compression benchmark, as the prediction.

We consider three different Look-up Table Models:

1) The Average Look-Up Table (AverageLT): This model uses the average latency of the packets that travel through the switch as a metric to summarize network's usage. As

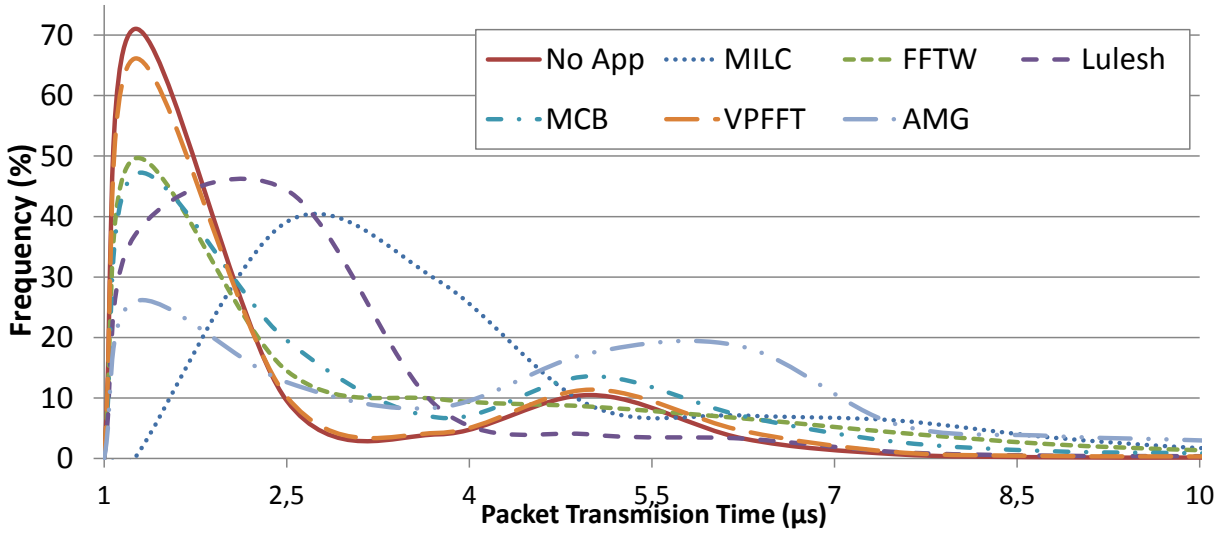

Fig. 3. Distributions of Packet Latencies on Cab

such, to predict the slowdown of application A when co-runs with application B, the model takes the average latency of the packets triggered by B, μ_B, which can be computed via impact measurements, and then looks for the input configuration C_i of the compression benchmark that has the closest average value μ_{C_i}, also computed via impact. Once this identification is done, the model takes the slowdown that application B suffers when is co-executed with the selected compression workload.

2) The Average and Standard Deviation Look-Up Table (AverageStDevLT): This model works in a very similar way as the previous, but instead of using just the average to select the input configuration of the compression benchmark to be used to predict, it uses the average and the standard deviation. As such, it takes the interval $I_B = [\mu_B - \sigma_B, \mu_B + \sigma_B]$ and the intervals $I_{C_i} = [\mu_{C_i} - \sigma_{C_i}, \mu_{C_i} + \sigma_{C_i}]$ for all the input configurations C_i of the compression benchmark, computes the lengths of the intervals $I_B \cap I_{C_i}$ and selects the configuration C_i that maximizes it. The idea behind this approach is to use the intervals I_{C_i} as proxies of the whole distribution of packets' latencies.

3) The Probability Distribution Function (PDF) Look-Up Table (PDFLT): This model works very similarly as the previous ones, but it uses the whole distribution of latencies instead of just the average and the standard deviation. As such, if the application A runs with B, the model takes the distribution of the packet latencies triggered by B, f_B and the distributions f_{C_i} of all the considered compression workloads. Then, it computes the integrals $\int_0^\infty f_B f_{C_i}$. Since we have that $\int_0^\infty f_B f_{C_i} \leq \int_0^\infty f_B \int_0^\infty f_{C_i} \leq 1$, these integrals are well defined. Since the closer distributions f_B and f_{C_i} are, the bigger the integrals' values are, the model selects the configuration C_i that maximizes $\int_0^\infty f_B f_{C_i}$.

B. Queue Theoretic Switch Metric

While packet latency distributions can provide some insight into the effective capability of the switch, they do not vary monotonically with application performance since it is not clear whether one distribution represents more or less network utilization than another (e.g. compare Lulesh and MCB's distributions). However, they can be used to extract the appropriate metric by modeling the behavior of a switch as a mathematical queue and leveraging the results of queuing theory (QT) [27] to infer the state of this queue based on its observable behavior (the packet latencies).

We represent the real switch as a queue by considering that each packet arrives at one switch port, is processed by internal switch circuitry and then departs via another port. As Figure 1 illustrates, when the packet arrives at this queue other packets may already be waiting in the queue to be routed, forcing the packet to wait until these packets are processed. The length of the queue inside the switch depends on the pattern of packet arrival times at the switch. Specifically, we use the M/G/1 [20] queue model to represent switch routing logic.

QT defines the utilization of a queue as the proportion of its entries that are used by the arriving traffic. Utilization ρ can be expressed as the rate $\frac{\lambda}{\mu}$, where λ is the mean rate of packet arrivals and μ is the mean rate of packet service times. If $\rho \geq 1$ then the queue's waiting time will grow, which implies that the switch will be contended and application performance will degrade significantly. Parameters λ and μ must be known to measure ρ. μ is a hardware parameter that is measured by sending multiple individual packets into an idle switch and measuring their minimum latency. λ is an application specific parameter that can only be directly measured by using switch counters, which are not available in general as they require root privileges. However, λ can be computed via the the Pollaczek-Khinchine formula [12]:

$$W = \frac{\rho + \lambda \mu Var(S)}{2(\mu - \lambda)} + \mu^{-1} \quad (1)$$

Where W is the total average time spent by packets in the queue either waiting and being serviced and $Var(S)$ is the variance of the service times. Since utilization $\rho = \frac{\lambda}{\mu}$, we can

write the formula as:

$$W = \frac{\frac{\lambda}{\mu} + \lambda\mu Var(S)}{2(\mu - \lambda)} + \mu^{-1} \quad (2)$$

which can be transformed to compute λ as follows:

$$\lambda = \frac{2 - 2W\mu}{-2W + \frac{2}{\mu} - \mu Var(S) - \mu^{-1}} \quad (3)$$

$Var(S)$ can be computed from the single-packet experiments on an idle switch and importantly, W is just the average latency of the packets communicated by ImpactB while the target application runs. Since utilization $\rho = \frac{\lambda}{\mu}$, we can compute it by using the the above formula given the parameters obtained through ImpactB measurements.

C. Switch Utilization of *CompressionB*

To quantify the fraction of switch capability that various configurations of CompressionB use, we run it together with ImpactB just like any other software component ImpactB may measure. This measurement makes it possible to relate performance degradation to the fraction of switch queue capability removed by CompressionB. The result is a high-level description of application performance in terms of a generic measure of network capability, the queue utilization fraction.

Our CompressionB+ImpactB experiments are executed using the same configuration as above, where we map 1 ImpactB and 1 CompressionB process on each socket, for 2 ImpactB and 2 CompressionB tasks per node. Figure 6 shows the range of different queue utilization percentages that can be achieved by all the considered variants of CompressionB when run on Cab. Parameter P, the number of partner processes, takes values 1, 4, 7, 14 and 17. Parameter B, the number of cycles the benchmark sleeps, has values $2.5E4$, $2.5E5$, $2.5E6$, $2.5E7$. Finally, parameter M, the number of messages sent in each round of communication, is either 1 or 10. As such, we consider 40 different input configurations of CompressionB.

The data shows that main determinant of switch queue utilization is the number of cycles the benchmarks sleeps, with utilization decreasing with longer sleeps. Further, utilization rises with increasing partner counts and message counts. The effect of partner count is strongest for longer sleep times while the effect of message count is strongest for shorter sleep times. In total, we consider 40 different input configurations, which allow us to cover switch queue utilization between 26% and 92%. The broad range of queue utilization provided by these configurations enables us to precisely evaluate applications performance degradations due to reduced switch capability.

D. Application performance impact due to reduced network capability

We used CompressionB to measure the relationship between available network switch capability and the performance of our target applications. Each experiment used the same configuration as in Section III-A, with 2 CompressionB processes per node. We assigned 1 CompressionB process per socket. The other cores were assigned to the application or left idle. Figure 7 shows the percentage performance degradation on Cab of FFTW, Lulesh, MCB, MILC, VPFFT and AMG (y-axis) as the percentage of switch utilized by CompressionB changes across its full range (x-axis) due to the use of different configuration parameters. Performance degradation is computed as $\frac{\text{Run time with interference - Run time with no interference}}{\text{Run time with no interference}}$. The left sub-figure shows the data with a linear y-axis and the right sub-figure uses a logarithmic y-axis. For each application we fit the data points with the best linear approximation to highlight the overall trend of the results.

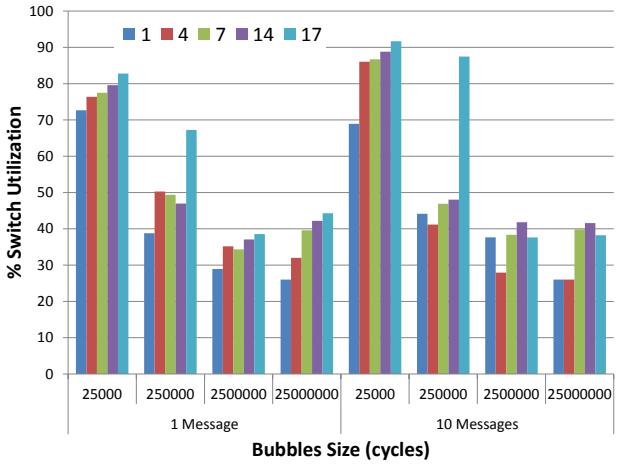

Fig. 6. Switch usage compression benchmark on Cab. We consider compression workloads with 1, 4, 7, 14 and 17 partners.

Reducing switch capability has the most effect on FFTW and VPFFT. FFTW runs more than 50% slower on Cab when even 40% of the switch queue is utilized and up to 250% slower as utilization reaches 92%. VPFFT also shows a very significant performance degradation, reaching a slowdown higher than 250% when 87% of the queue is used. VPFFT behavior is not as consistent as the ones observed in the other applications, showing oscillations from 132% to 263% of slowdown when 87% of the switch is used. MILC is also significantly affected, running approximately 20% more slowly on Cab at 40% queue utilization and over 100% more slowly at 92% utilization. This is because both applications are very sensitive to the latency of messages, meaning that if on average the queue is 40% full the stochastic nature of packet arrivals means that there are many packets that arrive when the queue is very long. Recall that the packet latency distributions shown in Figure 3 have some high latency packets even when the switch is idle. When the switch is partially utilized the fraction of high latency packets can become considerable, significantly degrading the performance of FFTW, VPFFT and MILC.

In contrast, Lulesh, MCB and AMG are significantly less affected by reductions in switch capability. The performance of Lulesh degrades by 8% at 50% queue utilization and 15% at 92% utilization. MCB and AMG are almost completely insensitive to queue utilization, slowing by no more than 3.5% across the full utilization range.

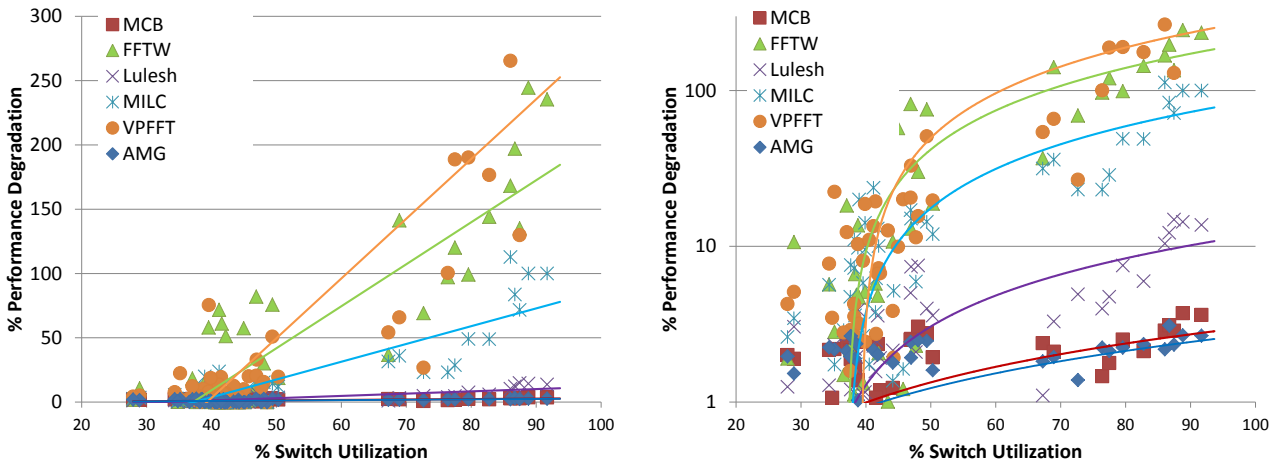

Fig. 7. Performance degradations suffered by real application in terms of switch queue utilization by CompressionB availability in cab. In the left, the y-axis is expressed as a linear scale. In the right, it is expressed as a logarithmic scale.

The above experiments make it possible to estimate the performance of software components when executing on switches with different capabilities. Specifically, to focus on a particular scenario it is necessary to choose the queue utilization fraction that corresponds to the removal of the given amount of switch capability and run the application with CompressionB configured to emulate this utilization fraction.

V. Prediction

Section III-A shows how to measure the application's utilization of switch resources by timing the latencies of individual packets (the ImpactB benchmark). Section III-B then presents a way to emulate switches with reduced capabilities by running concurrently with the application an interference workload (the CompressionB benchmark) that reduces the amount of switch capability available to the application. Section IV describes several methodologies to predict applications' slowdowns when network resources are not fully available by combining measurements taken by using the ImpactB and the CompressionB benchmarks. Further, IV describes a way to quantify the amount of interference induced by CompressionB in terms of a queue utilization metric.

We now show how to combine these two experimental techniques to make quantitative predictions about how the performance of multiple software components (application tasks or entire applications) will suffer when they are executed concurrently on the same switch. Critically, our approach makes it possible to make predictions for new combinations of software components (number of combinations grows exponentially with the number of components and polynomially in the number of their configurations) based on experiments performed on each component in isolation from the others (grows linearly with the number of components).

We evaluate the accuracy of the proposed prediction algorithms by running pairs of our target applications concurrently on the same switch to observe whether the model correctly predicts how much they degrade each other's performance. We run each benchmark in continuous loops and we measure the average slowdown over many concurrent runs. In these experiments each application is executed using the configurations used in the experiments reported in Sections III-A and III-B. Specifically, for the experiments run with MILC, FFTW, MCB, AMG and VPFFT we ran 4 processes on each socket, for a total of 144 processes on the 18 dual-socket nodes connected to one switch. Since Lulesh must run on cubic numbers of processes, we ran 2 Lulesh processes on each socket on 16 nodes, for a total of 64 processes. This process mapping utilizes at most half the available cores, leaving enough cores for two applications to run concurrently without sharing cores. Our experiments include combinations where two copies of a single application run concurrently on the same nodes and switch, as well as combinations where two different applications execute together. The former evaluates our model's accuracy on the use-case of HPC capability computing where different amounts of a single application's work may be assigned to a single switch. The latter accounts for the use-cases more typical in cloud computing or HPC capacity computing where multiple applications may share a single switch, as well as applications that run processes dedicated to molecular dynamics and processes for FFT computations concurrently on different nodes on the same network. In table I we depict the measured slowdowns for all the possible application pairs. In each row we show all the possible performance slowdowns each applications experiments when is co-run with itself and with the other 5 applications. Numbers represent percentages.

	FFT	Lulesh	MCB	MILC	VPFFT	AMG
FFT	45	5	3	11	12	7
Lulesh	5	5	3	6	2	3
MCB	3	5	4	7	5	6
MILC	25	12	1	4	3	14
VPFFT	9	0	2	5	7	2
AMG	0	5	4	5	3	4

TABLE I
MEASURED PERFORMANCE SLOWDOWNS FOR ALL THE COMBINED WORKLOADS. NUMBERS EXPRESS PERCENTAGES.

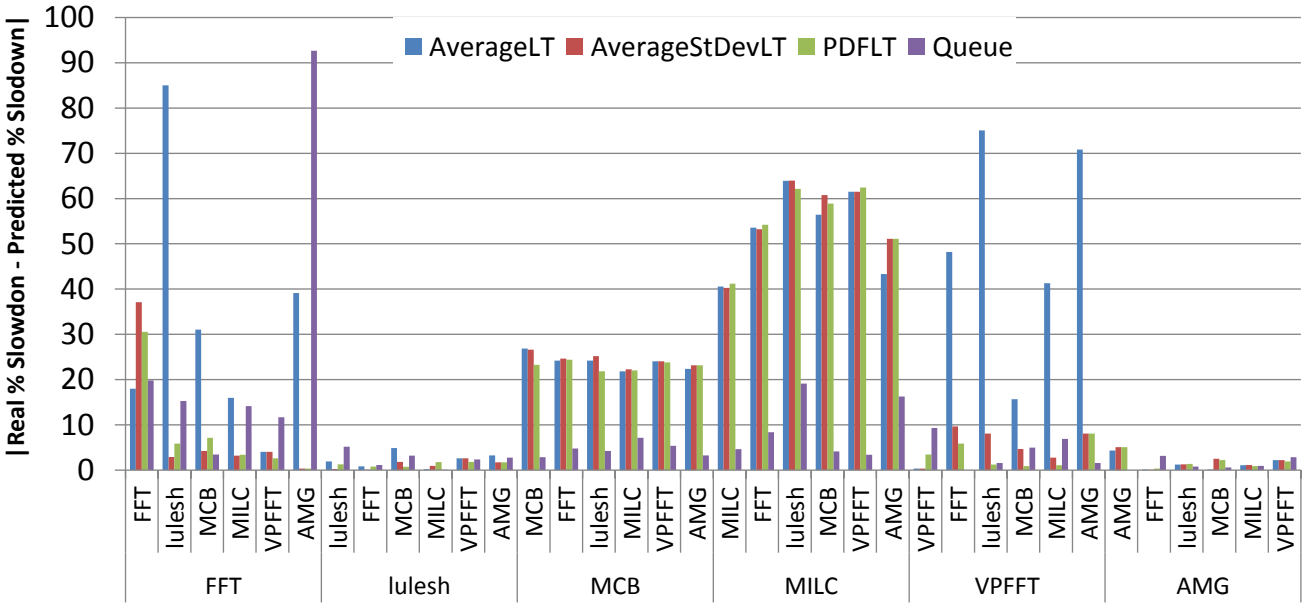

Fig. 8. Performance predictions for combined workloads in Cab

A. Look-up Table Predictions

Using the methodologies explained in section IV-A we get some predictions of the measured performance slowdowns when the considered applications share the network resources. To apply these methodologies, we run each one of the applications with all the 40 input configurations of CompressionB considered in section IV-C. Besides that, we run each one of these 40 configurations with the ImpactB to figure out, for each input configuration, the average packet transmission latency, the standard deviation and the complete distribution of the packets latency. With these data we can apply the three methodologies explained in section IV-A: AverageLT, AverageStDevLT and PDFLT. As such, if we want to predict the performance of application A when it runs with application B, we compute the closest configuration of ImpactB to application B and use the measured slowdown when A run with it as a prediction.

Figure 8 presents the results of the 36 experiments (6 experiments where each application was run with itself and 30 experiments for different application pairs) executed on Cab. The y-axis shows the difference between the measured and the predicted percent performance degradations of each application in each pairing, while the x-axis shows each pairing $X - Y$. Since experiments where two different applications are executed concurrently result in two different performance degradations, they are listed separately on the x-axis, for a total of 36 different degradation measurements. The x-axis is divided into 6 different boxes: Each box contains data referring to the application written horizontally when co-run with each one of the applications listed vertically.

Figure 8 shows the results we achieve using AverageLT, AverageStDevLT and PDFLT. The accuracy is quite good when the techniques predict slowdowns for lulesh and AMG. The averageLT model shows high deviations when it tries to predict slowdowns for FFT and VPFFT. However, the other two techniques improve averageLT accuracy when predicting the behavior of these two applications. For MILC and MCB none of the techniques considered in this section, AverageLT, AverageStDevLT and PDFLT, achieve good results, which means that we have to increase the number of ImpactB configurations we consider to build our look-up tables or either apply more complex models. In the next section, we discuss the prediction methodology and the results achieved when the queue model is used.

B. Queue Model Predictions

Using the model explained in section IV-B, we can measure the fraction of the switch queue that software components A and B use by conducting impact experiments on A and B. They will result in quantities $U_A\%$ and $U_B\%$ that measure the fraction of the switch queue each component utilizes. Compression experiments on these components produce mappings p_A and p_B that map queue utilization fractions to the performance degradation in each component, like we show in figure 7. We then use the configurations of CompressionB that also utilize $U_A\%$ and $U_B\%$ of the switch queue to model the effects of A and B, respectively on other software components with which they share a switch. We thus predict the performance degradation of A when executed concurrently with B to be $p_A(U_B)$. Specifically, this means that A's performance will degrade as much when sharing the switch with B as it did when it shared the switch with the configuration of CompressionB that utilizes the same fraction of the switch queue as B does. The converse prediction is made for B. This

analysis can be performed for any combination of application tasks, their configurations (e.g. number of molecules simulated or the size of their communication stencil) or even multiple concurrently executing applications.

Results in figure 8 show that overall the queue model has very good predictive capability. For lulesh, MCB, VPFFT and AMG the model properly predicts the performance slowdowns for all the possible co-running applications, as it clearly separates the pairings that induce little performance degradation from those that induce significant degradation. For MILC and FFTW, the model shows some deviations sometimes, which can be divide into three different categories: (i) the model predicts zero degradation while in reality performance degrades by 3%-5%, (ii) it predicts a degradation that a few percent higher or lower than reality (MILC with Lulesh and MCB) or (iii) the model predicts a notable degradation where in reality it was small (FFTW with AMG).

The only significant error is when the model predicts the performance of FFTW when co-executing with AMG. According to the model, the performance of FFTW would degrade significantly more than it actually does. The explanation of this high error is that, as AMG executions go through phases that do not significantly use the network, the switch capacity available to FFTW is close to 100% during a significant portion of its co-run with AMG, which is something that the queue model has not considered as it assumes a constant utilization of the network during the applications' runs.

C. Summarization of the Results

In figure 9 we summarize the results we have obtained with the four considered methodologies. For each method, we show two boxes that represent the second and third quartiles of the errors we have got predicting the 36 considered workloads. The line between the two boxes represents the median and the two error bars show the range covered by the errors of the first and the fourth quartile respectively. As we can see, the AverageStDevLT model outperforms the AverageLT, which is not surprising since the former uses more data than the latter. The accuracy of models AverageStDevLT and PDFLT is almost the same, which means that just the average and the standard deviation of the packets' latency is already a good description of the whole distribution of latencies and that adding more information regarding this distribution does not increase the accuracy of the models. However, look-up table approaches do not achieve a satisfactory accuracy as more than one third of the predictions have an accuracy worse than 20%.

The queue model over-performs in general the look-up table approaches as more than 75% of its predictions have an error lower than 10%. Even more, as we can see in figure 8 all of its predictions except one have an error lower than 20%. However, the queue model still shows a high deviation in one of the considered workloads.

VI. Related Work

The importance of network performance optimization has motivated significant research by the performance analysis

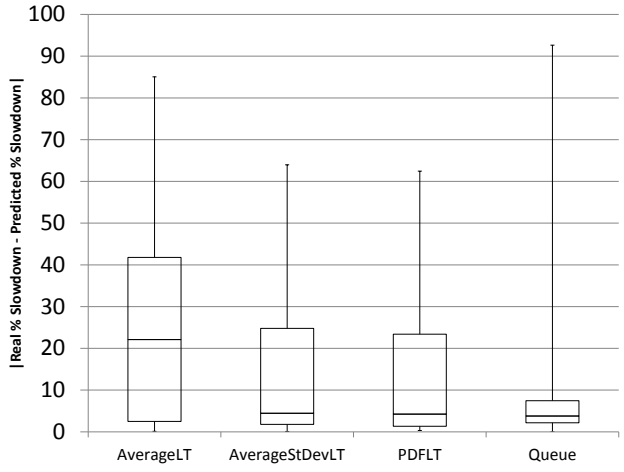

Fig. 9. Summarization of the results obtained in Cab

community. It can be divided into two categories: simulation and indirect measurement. The simulation approach, exemplified by tools such as SST [22], BigSim [29], Dimemas [18] or Venus [23] uses a detailed model of network hardware to account for the path of every message sent by each application node. Although these tools can accurately predict the performance of a particular application configuration on a particular network design, they have two limitations. First, the cost of using them can be high for many realistic large-scale applications since a full analysis requires a large-scale application run followed by a detailed simulation of its communications and, second, each simulation is valid for only one application configuration.

The indirect measurement approach is exemplified by tracing tools such as Vampir [15] and Paraver [17] as well as performance counter-based tools such as Tau [25]. In this approach various application regions are monitored to determine its communication structure, the amount of time it spends performing various operations and the number of events such as cache misses that occur during each operation. While these measurements can be used to derive non-trivial information of HPC applications, like the internal structure of their executions [6] or the reasons behin performance slowdowns slowdowns [5], they can only enable indirect inference about how the properties of a network relate to application performance.

Ohter work has been more focused on exploring the usefulness of new network topologies to achieve significant reductions in both network cost and network power, while still providing a balance of high global and high local bandwidth [28]. The idea is to reduce the length of network links to get both lower cost and lower power while keeping a good throughput. This is done by developing new network topologies with special properties in terms of connectivity.

Previous work [21] considered how power may reduce interconnection networks' capacity. There have been several prior efforts to reduce power in interconnection networks with a particular focus on reducing power on infrequently used links.

Both [14], [26] propose techniques to power down certain links in response to traffic behavior. The techniques presented in this paper can be used to evaluate this kind of power optimizations for interconnecting networks.

Measurements of network dynamics have been obtained through active measurement techniques [4]. Since it is impractical to monitor every link on an end-to-end path, the authors inject multicast traffic to infer network-internal characteristics. Other sophisticated and effective algorithms based on active-measurement techniques have been derived lor large-scale network tomography [9]. In particular, these active measurement techniques developed in the contect of network tomography have been used to study the heterogeneous and unregulated structure of internet [8].

VII. Conclusion

In this paper we have shown the usefulness of proactive measurements to analyze applications' consumption of switch resources and to predict performance degradations when those resources are shared with other workloads. This a very important problem since high performance computing infrastructures typically run several applications on the same time, all of them sharing the network. Our technique uses two interferences that inject extra network workload. The first detemines the fraction of the network that is utilized by a software component (an application or an individual task) to figure out the existence and severity of network contention. The second aggressively injects network packets while a software component runs to evaluate its performance on networks with less capacity or when it shares network resources with other software components.

We then combine the information from the two types of experiment to predict the performance slowdown experienced by multiple software components (e.g. multiple processes of a single MPI application) when they share a single network. We have also validated our approach by comparing the predictions we get through our modeling and measurement techniques with real measurements obtained when two applications run together on the same switch. By using a queuing theory based approach we have been able to achieve excellent accuracy in almost all the considered workloads. Also, our methodology is general in the sense that can be deployed in any kind of HPC infrastructure that uses any kind of interconnecting network to handle communication between computing nodes.

References

[1] Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory.

[2] *Scientific Application Performance on Candidate PetaScale Platforms*. IEEE Computer Society, 2007.

[3] G. Bauer, S. Gottlieb, and T. Hoefler. Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application su3_rmd. In *CCGRID*, pages 652–659, 2012.

[4] R. Caceres, N. Duffield, J. Horowitz, and D. Towsley. Multicast-based inference of network-internal loss characteristics. *Information Theory, IEEE Transactions on*, 45(7):2462–2480, 1999.

[5] M. Casas, R. M. Badia, and J. Labarta. Automatic analysis of speedup of mpi applications. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 349–358, New York, NY, USA, 2008. ACM.

[6] M. Casas, R. M. Badia, and J. Labarta. Automatic phase detection and structure extraction of mpi applications. *Int. J. High Perform. Comput. Appl.*, 24(3):335–360, Aug. 2010.

[7] J. Cetnar, J. Wallenius, and W. Gudowski. MCB: A Continuous Energy Monte-Carlo Burnup Simulation Code. In *Actinide and Fission Product Partitioning and Transmutation*, 1999.

[8] M. Coates, A. Hero, R. Nowak, and B. Yu. Internet tomography. *Signal Processing Magazine, IEEE*, 19(3):47–65, 2002.

[9] N. Duffield and F. Lo Presti. Multicast inference of packet delay variance at interior network links. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1351–1360 vol.3, 2000.

[10] R. D. Falgout and U. M. Yang. hypre: a library of high performance preconditioners. In *Preconditioners, Lecture Notes in Computer Science*, pages 632–641, 2002.

[11] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] J. Haigh. *Probability Models*. Springer, 2002.

[13] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[14] E. J. Kim, K. H. Yum, G. M. Link, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, M. Yousif, and C. R. Das. Energy optimization techniques in cluster interconnects. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 459–464, New York, NY, USA, 2003. ACM.

[15] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir Performance Analysis Tool-Set. In M. M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Parallel Tools Workshop*, pages 139–155. Springer, 2008.

[16] P. Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA IPTO, September 2008.

[17] J. Labarta. New Analysis Techniques in the CEPBA-Tools Environment. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Parallel Tools Workshop*, pages 125–143. Springer, 2009.

[18] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A Parallel Program Development Environment, 1996.

[19] R. A. Lebensohn, A. K. Kanjarla, and P. Eisenlohr. An elasto-viscoplastic formulation based on fast fourier transforms for the prediction of micromechanical fields in polycrystalline materials. *International Journal of Plasticity*, 3233(0):59 – 69, 2012.

[20] M. F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications*, volume 5. 1989.

[21] G. Patel, S. Chai, S. Yalamanchili, and D. Schimmel. Power constrained design of multiprocessor interconnection networks. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pages 408–416, 1997.

[22] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, Mar. 2011.

[23] G. Rodriguez, R. Beivide, C. Minkenberg, J. Labarta, and M. Valero. Exploring Pattern-aware Routing in Feneralized Fat Rree Networks. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 276–285, New York, NY, USA, 2009. ACM.

[24] V. Sarkar. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Technical report, DARPA IPTO, September 2009.

[25] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[26] V. Soteriou and L.-S. Peh. Design-space exploration of power-aware on/off interconnection networks. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 510–517, 2004.

[27] V. Sundarapandian. *Probability, Statistics and Queueing Theory*. PHI Learning, 2009.

[28] K. D. Underwood and E. Borch. Exploiting Communication and Packing Locality for Cost-Effective Large Scale Networks. 2012.

[29] G. Zheng, G. Kakulapati, and L. V. Kalé. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *IPDPS*. IEEE Computer Society, 2004.